Project Integration Architecture: Formulation of Semantic Parameters

Dr. William Henry Jones
National Aeronautics and Space Administration
John H. Glenn Research Center at Lewis Field
Cleveland, OH 44135
216-433-5862
William.H.Jones@grc.nasa.gov

```
X00.00  13 Jan 2000
X00.01  03 Mar 2000
X00.02  19 Sep 2000
```

Keywords:
PIA; PRICE; CORBA; Knowledge Management; Application Integration;
Technical Integration Technologies; Semantic Encapsulation; Semantic Infusion;

ABSTRACT: *One of several key elements of the Project Integration Architecture (PIA) is the intention to formulate parameter objects which convey meaningful semantic information. In so doing, it is expected that a level of automation can be achieved in the consumption of information content by PIA-consuming clients outside the programmatic boundary of a presenting PIA-wrapped application. This paper discusses the steps that have been recently taken in formulating such semantically-meaningful parameters.*

1 Introduction

The analysis of the whole of an engineering system frequently involves a number of cooperating analyses, each focusing on a particular discipline of analysis relevant to the whole. Each of these analyses, in turn, deals with a number of parameters characterizing the particulars of the analysis. Traditionally, the meaning of these parameters is 'understood' only within the application code in which they are found. This understanding is in the form of program statements which consume each particular parameter as an input to a computation, or generate the parameter as a result of a computation.

To another analysis, such a parameter is only a number without meaning, at least until such time as a user of that other analysis supplies the parameter as an input in some particular spot. By so doing, semantic meaning is attached to the number, but that understanding is, again, by the nature of the coding in which the input is used. The consuming analysis is not able to look at the number an say "Ah, a viscosity. Just what I need.", but is instead told "Here is a number. Use it as the viscosity number in the program."

The nature of this traditional method for establishing the semantic meaning of a parameter results in the persistent need for something (usually a person) to arrange things. It is this need that so often hampers the arrangement of cooperating analysis codes into, well, cooperating analysis codes. It is often the case that the cooperation is between the people tending the codes rather than between the application codes themselves. Not infrequently this cooperation devolves to the most cumbersome, error prone, manual forms imagineable.

Of course, many worthy attempts have been made to treat the topic of code cooperation, usually through the vehicle of a well-known file format to be accepted and supported by all of the codes of a cooperating suite. In such approaches, the semantic meaning of a parameter is established by its location within the file structure. For instance, in a simple file structure, it might be declared that the sixteenth number in the file is always the exit total pressure of the flow field described by the file, whatever that flow field might be. Such a solution to the problem is, of course, entirely valid, but for mechanical reasons the approach has not achieved widespread acceptance.

In dealing with this area, the Project Integration Architecture (PIA) [1] takes a different approach by capitalizing upon the object technology with which it is implemented. Objects are, to begin with the abstract, programming entities that have functionality and state. That is, objects have data and they will do things to and based on that data. Objects, further, are of distinct kinds which can be distinguished as a programming act.

Given this first basis of object technology, it is then a natural extension to consider object kinds as being or representing particular things; this kind of object is a printing device while that kind of object is a bank account. By this simple step, semantic meaning is now attached to the object based upon its distinguishable kind. Objects of this particular kind are understood as bank accounts and the number they yield up is, obviously, the current balance in the account.

It is precisely this capacity of objects which PIA uses to establish the semantic nature of parameters. An object of this particular kind is understood to be a gas total pressure and, thus, the number it provides when requested is not just a number, but a gas total pressure number. By this formulation, PIA establishes the semantic nature of any given number not by its position in a file or a structure or an input stream, but by the kind of object in which the number is presented. Thus, a total pressure is a total pressure without regard to where it is found.

2 Substantive Details

2.1 Basic Parametric Object

As indicated in the introduction above, PIA establishes the semantic nature of parameters by means of the object kind in which any given parameter is presented. This choice, far from being a lightning stroke of technological innovation, is nearly dictated by the nature of the object-oriented technology within which PIA is being implemented. It flows naturally from the concept of an object kind and from the mechanisms which allow object kinds to be distinguish in the course of program operation.

PIA uses the further concept of object inheritance in this semantic use. In many object-oriented technology systems (including those used to implement PIA), object kinds (or, more correctly in C++ terminology, the classes of which the objects are instances) may be derived from other, base object kinds and, in so doing, inherit the characteristics of the base kind. (In some object-oriented technologies, object classes may be derived from more than one class, inheriting the characteristics of all the base classes; however, the PIA project, as a choice, has adhered to a strict single-inheritance design.) This derivation and inheritance may proceed on layer by layer, deriving kind from kind, to any practical depth.

Using these object concepts, PIA first declares that there is an object kind that is, simply, a parameter. At this level, a parameter is considered to be some as yet unspecified entity defining in part the state of an application. The combination of the complete set of parameters of an application defines a unique state for that application.

The parameter object, as mentioned above, does not yet specify what its content is. Nevertheless, a number of useful characteristics common to all parameters are defined at this level. Among other things, a parameter may be noted to be either an input or an output of the analysis (or both if appropriate, and possibly neither), and it may participate in a parametric dependency graph in which changes in one parameter may necessitate changes in one or more other parameters.

Another small contribution at this level is the implementation of change history mechansisms. While the exact nature of a parameter's content is still completely undefined, it is presumed that its state can be represented as a text. Mechanisms are provided to capture, date, and record such a text as representing the prior state of a parameter at the time of some significant change. Thus, all parameter objects, of whatever type, have available a mechanism for tracking their history through the engineering process.

2.2 Structural, Atomic Forms

With the parameter object kind as a basis, PIA then proceeds to specialize that kind by declaring derivatives to represent scalar, vector, and matrix parameters. At this level the question as to just what is arranged into scalar, vector, or matrix form is still open, but the structural arrangement of information within the parameter is now clear. Functionality is provided by each object type to facilitate access to data within its structural form.

Another structural form is derived from the foundational parameter level: the organizational parameter. The general outlook of the organizational parameter is that it contains no directly consumable data (that is, it has no pressures or velocity vectors or the like), but instead contains information and structure which organize other parameters into a whole. The advantage of the organizational parameter is two-fold: it avoids a protocol-based constraint upon the parametric identification system of the application architecture (again, see [1]) and it adds a considerable programmatic flexibility for parameter organization and navigation that does not exist in the static identification system.

Specialization of the scalar, vector, and matrix parameter forms proceeds with another layer of derivation to provide the various atomic data types common to engineering analysis. Boolean, **long** (integer), **double** (floating point), and string forms are derived (as appropriate) for the scalar, vector, and matrix foundations.

2.3 Measurement Forms

The next derivational layer adds to **double** parametric forms the various concepts or metrics of measurement [2]. That is, first a form declaring that dimensionality exists is defined, followed by object types based upon that form that encapsulate the concepts of length, time, velocity, mass, temperature, and the like. A nondimensional form explicitly declaring that the encapsulated parameter does not have a dimensional characteristic is also defined as, paradoxically, a derivative of the dimensional form.

The dimensional base form provides functionality to convert the encapsulated measurement between various systems of measurement. The derived dimensional forms need only add particulars as to the dimensional nature of their form. Thus, given, say, a force object, forces are converted between pounds, Newtons, and dynes simply by chosing the correct access functions. In the case of the nondimensional form, the particulars added are that there is no actual dimensional nature to the form; however, by so doing, non-dimensional parameters may be freely intermixed with dimensional forms in computations.

The conversion of measurement systems is considered a helpful, if not entirely revolutionary, innovation. Code that acquires information from such parameters can achieve insensitivity to the actual measurement system of the parameter by the simple act of requesting information in the desired measurement system. Further, code may be written that does not explicitly know what system of measurement it is working in by asking the host object for a code number specifying the measurement system and supplying that returned code to other parameters contributing to the computation.

Proceeding beyond this measurement system layer, derivation of parameter forms moves to physical types. Things such as gas static and total pressures, enthalpies, Mach and Reynolds numbers, and the like are defined. Derivational layering is still present to accomodate specializations even of these forms. For instance, Mach number can be specialized to forms such as the far-field Mach number, the local Mach number, and, again, the like.

2.4 Ancillary Information

The establishment of semantic type by object kind described above is not always sufficient to establish the use (or non-use) of a particular parameter. For instance, in the presense of multiple instances of a particular parameter kind, some further discriminator may be needed to identify which parameter of the set is the appropriate one. Unfortunately, such selections are usually specific to the needs and semantics of the situation and no single mechanism can be defined which provides a universally useful approach.

One factor that is often important in such situations is the physical location of the information encapsulated by the parameter. That is, where in the geometrical space of the engineering system does this parameter apply. For example, the total pressure of the gas flow at the exit of a jet engine fan assembly is of particular interest to a compressor analysis while the total pressure of the flow at the fan entrance is of little relevance.

To provide this sort of information, PIA provides a positional description form as a part of its larger attached descriptive mechanism. Each application object (of which parameter objects are, themselves, a kind) can attach one or more of a wide set of descriptive forms at each derivational level of the application object kind. By making use of this mechanism and attaching a positional description to a parameter, especially when many parameters of the same kind coexist, a mechanism of discrimination may be provided.

The positional description is, naturally, not the only type of ancillary information that can, or could be, added. A propulsion station description has been devised, as well as a deliberately nebulus related parameter mechanism. Indeed, the limits of invention are the only practical frontier for such mechanisms.

2.5 Contextural Information

A further opportunity to infer the semantic content of parameters exists in the arrangment of applications into directed graphs (dicussed in [3]). By existing in an application which is a predecessor, whether immediate or extended, of a consuming application, a parameter asserts a basic semantic relevance to that consuming applicaton.

As an example of such contextural semantics, consider a graph of applications which analyze the gas flow through the various components of a jet engine. Presumably, such a graph would be arranged much in the manner of the actual flow: inlet connected to fan, fan connected to compressor and to fan duct, compressor connected to burner and to customer bleed, and so on. In such a graph, the compressor flow analysis would consider parameters in the fan flow analysis to be of semantic relevance because of their existence in a predecessor application, while it would consider parameters of the same kind existing in the combustor analysis to be of little or no relevance because of their existence in a successor application.

Another aspect of such contextural inference (again, discussed in greater detail in [3]) is some sense of 'closeness' within the application graph. Parameters that are more immediate to a consuming application within the metrics of a directed graph of cooperating applications may well be more meaningful in a semantic manner than those of like kind existing at some greater remove. Continuing the jet engine flow example, parameters of the compressor analysis are probably more meaningful to the burner analysis than are parameters of similar kind in the fan analysis.

3 Implementation Options

The implementation of semantic information discussed above, in particular the declaration through derivation of kind, is clear and straightforward. The approach builds upon object discrimination mechanisms that are in place and well established. Unfortunately, the astute will notice (and probably point out), that the single-inheritance dictum adopted by PIA has introduced a certain amount of duplication into the object kind implementation. In particular, once the great foundations of scalar, vector, and matrix forms are set, common concepts such as systems of measurement must be replicated in each foundation.

The confluence of separate base concepts in singly inherited derivational systems is a point of difficulty. When one conceptual system is to be widespread throughout a system of object kinds, it is quite easy to encapsulate that conceptual system in a base class and, through derivation, allow it to be inherited throughout all of those object kinds. When two such conceptual systems exist, but their consuming object kind sets are not identical, no such easy solution exists.

In this case, parameters may be scalars, vectors, or matricies and they may or may not be dimensional in any one of a number of dimensional forms (length, velocity, mass, non-dimensional, etc.). The difficulty arises in deciding which of these conceptual systems should be foundational. Should scalars, vectors, and matricies each be specialized into various dimensional forms, or should the plethora of dimensional forms each be specialized into scalars, vectors, and matricies?

As discussed above, the selection made by PIA (for the moment) is to make the more complex concept (the structural forms of scalar, vector, and matrix) foundational and to specialize them with the less complex and more easily replicated dimensional concept. This is not an uncommon approach and, indeed, is also used here in specializing scalars, vectors, and matricies into scalars, vectors, and matricies of Booleans, **long**s, **double**s, and strings. The further specialization of object kinds to systems of measurement gains some additional justification since it is applied, in fact, only to scalars, vectors, and matricies of **double**s since only a form capable of representing a continuum was considered appropriate to systems of measurement.

Despite all of this fine-sounding language, there is still the small pain that alarms each time one finds oneself replicating the same form of code or object over and over and over. This is, to some extent, the antithesis of object orientation. Things worth doing should only be done once and should be inherited (and perhaps bent a little bit when necessary) through the mechanism of derivation. As a consequence, considerable thought has been given as to what else might have been done.

3.1 Multiple Inheritance

Some object technologies, for instance that of C++, provide for multiple inheritance in derivation. One could declare one set of objects that are scalars, vectors, and matricies and another set of objects that are lengths, velocities, masses, etc., and then derive a particular object based upon the vector and length kinds, thus inheriting both those characteristics.

The difficulty with systems such as this is that 'things' can become quite complicated, even when compared to the fascinating complications of object-orientation itself. In particular, the matter of object kind discrimination becomes rather difficult. Multiple-inheritance means that, now, there are multiple answers (at a given level) to the question "What kind of object are you?"

Beyond this, multiple-inheritance confuses the issue as to which copy of common elements inherited by the multiple bases is ‘the’ copy. If vectors and lengths share some common heritage, which copy is used in an object that is both a vector and a length? Multiple-inheritance environments do provide answers to such questions, but such answers are usually not on the road to clarity.

Finally, the governing fact is that the PIA project has selected a dictum of strict single inheritance. A multiple-inheritance solution would require a complete reworking of the very foundations of the PIA implementation.

3.2 Merging the Concept of Dimensionality

Another alternative would be to merge the multi-faceted concept of dimensionality into a single construct and place it in the parameter foundation from whence it would be inherited by scalars, vectors, and matricies. Thus, instead of there being lengths, velocities, masses, *et al*, there would only be dimensional parameters whose dimensionality would be a discoverable characteristic. Such a condensed concept of dimensionality could then become a characteristic of the parameter base class and be inherited by scalars, vectors and matricies without the necessity of replicating these structural forms for every type of dimensionality.

Such a concept has considerable appeal and may well prove in the light of future experience to be a better choice. The difficulty seen at this juncture is that, by making dimensionality a characteristic of all parameters, it is effectively removed from the object kind mechanism. It was thought that the ability to interrogate a parameter object and ask, for instance, “Are you a kind of velocity?” was a substantial capacity, as is the ability to ask “Are you a kind of vector?” Making dimensionality a characteristic of parameters would have necessitated the invention of a second mechanism peculiar to parameters to answer the question “Are you a kind of velocity?” Such a maneuver was judged even more antithetical to object-oriented architecture than the replication of dimensionality in each of the structural forms.

Another difficulty with the approach was that, to properly form the dimensionally sensitive functionality, it would have had to have been declared in the parameter base class before it would have acquired meaningful definition. For example, the parameter scalar **double** class declares and implements a **GetDoub** function which returns the encapsulated **double** value. To be properly formed as a dimensionally sensitive function, the **GetDoub** function would have to be declared (and implemented in a non-functional manner) in the parameter base class before the concept of scalars and **double**s had been introduced. This, again, seemed antithetical to object-oriented design.

3.3 Tieing of Object Kinds

Another common technique in situations such as this is to tie objects of two foundational classes together through a pointer and pass-through functionality. Usually, the more complicated class implements a pointer to an instance of a less complicated class and declares and/or implements functionality which, in fact, is simply passed on to the instance of that less complicated class.

In this situation, dimensionality would probably be the less complicated class and it would be specialized through derivation into length dimensionality, mass dimensionality, velocity dimensionality, and the like. The parameter class would be implemented with a pointer to a dimensional object (which might be null for those parameter kinds which do not sustain the concept of dimensionality). Those parameters (the various **double** parameter forms) that have dimensionality would then realize an instance of the appropriate dimensionality kind and, as need arose, let that tied dimensionality object do what it will to the value encapsulated by the parameter object.

As a matter of fact, it was considered that such a scheme was very nearly in place in the form of the units description form of the application object layered descriptive system. The existing descriptive form provides a text annotating the units of the described object and a code value indicating the system of measurement used. (This is, in fact, where the implemented measurement derivational layer obtains its measurement system information.) It would only be necessary to further specialize the descriptive form through derivation to provide dimensional conversion (and any other dimensionality) functionality.

The arguments against this choice, though, are the same as before: by making the dimensionality an internal characteristic, the ability to discriminate parametric forms based upon the existing, cohesive object kind mechanisms is lost and, to the extent the functionality is desired, it must be implemented through a new mechanism peculiar to parameter objects.

4 Summary

The derivational specialization of parameter objects for PIA has been discussed and the infusion of semantic meaning into the parameter itself illustrated. This infusion of meaning is a change from previous approaches in which parameters obtained semantic meaning by virtue of their position in input/output streams and internal usage in a program.

A key benefit of semantic infusion through object derivation is that measurement types and systems of measurement can (and do) become encapsulated characteristics of parameter objects. Parameter objects can now ‘know’ whether they are in feet or meters, pounds force or Newtons and can, thus, tailor themselves through conversion to desired forms on demand.

References

[1] William Henry Jones. Project Integration Architecture: Application Architecture. Draft paper available on central PIA web site, March 1999.

[2] William Henry Jones. Project Integration Architecture: Formulation of Dimensionality in Semantic Parameters. Draft paper available on central PIA web site, March 2000.

[3] William Henry Jones. Project Integration Architecture: Inter-Application Propagation of Information. Draft paper available on central PIA web site, December 1999.